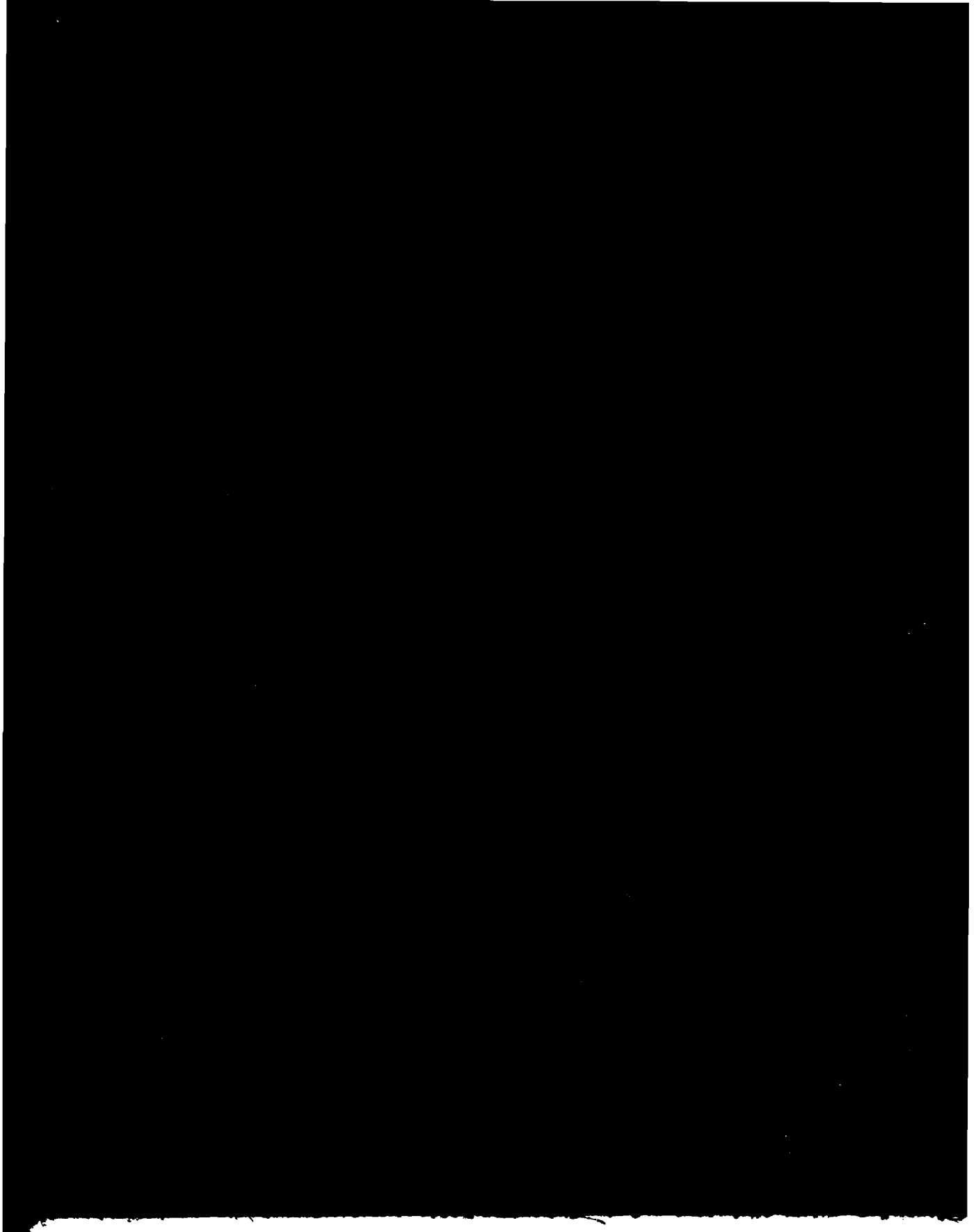
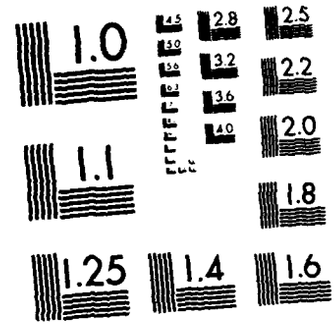
AD-A195 071 THE ROLE OF PROLOG (PROGRAMMING AND LOGIC) IN NATURAL LANGUAGE PROCESSING(U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY M L MCHALE MAR 88 RADC-TR-88-74 1/1

UNCLASSIFIED F/G 12/3 NL

END
DATE FILMED
8 88





RADC-TR-88-74
In-House Report
March 1988

THE ROLE OF PROLOG IN NATURAL LANGUAGE PROCESSING

Michael L. McHale

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-74 has been reviewed and is approved for publication.

APPROVED: Samuel A. D. Mitto

SAMUEL A. DINITTO, JR.
Chief, C2 Software Technology Division
Directorate of Command and Control

APPROVED: Raymond P. Urtz

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): RADC-TR-88-74	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: Rome Air Development Center; 6b. OFFICE SYMBOL (If applicable): COES	7a. NAME OF MONITORING ORGANIZATION: N/A
6c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	7b. ADDRESS (City, State, and ZIP Code): N/A
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable): COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N/A
8c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
62702F	5581	27	30

11. TITLE (Include Security Classification): THE ROLE OF PROLOG IN NATURAL LANGUAGE PROCESSING

12. PERSONAL AUTHOR(S): Michael L. McHale

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
In-House	FROM Sep 87 TO Dec 87	March 1988	40

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	05		Logic Programming; Definite Clause Grammars; PROLOG; CHAT-80; Natural Language Processing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report presents an introduction to the use of PROLOG for Natural Language Processing (NLP). First, some historical background information is covered, then, a review of some of the better known NLP systems in PROLOG including CHAT-80, the Bottom Up Parser (BUP) and Modular Logic Grammars is covered. Also included is a section on Definite Clause Grammars (DCG), the Japanese Fifth Generation Project, text generation, and the elements of good PROLOG style.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Michael L. McHale	(315) 330-2973	RADC (COES)

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The field of Artificial Intelligence strives to produce computer programs that exhibit intelligent behavior. One of the areas of interest is the processing of natural language. This report will discuss the role of the computer language PROLOG in Natural Language Processing (NLP) both from theoretic and pragmatic viewpoints.

The reasons for using PROLOG for NLP are numerous. First, linguists can write natural language grammars almost directly as PROLOG programs; this allows 'fast-prototyping' of NLP systems and facilitates analysis of NLP theories. Second, semantic representations of natural language texts that use logic formalisms are readily produced in PROLOG because of PROLOG's logical foundations. Third, PROLOG's built-in inferencing mechanisms are often sufficient for inferences on the logical forms produced by NLPs. Fourth, The logical, declarative nature of PROLOG may make it the language of choice for parallel computing systems. Finally, the fact that PROLOG has a de facto standard (Edinburgh) makes the porting of code from one computer system to another virtually trouble free. Perhaps the strongest tie one could make between NLP and PROLOG was stated by John Stuart Mill in his Inaugural Address at St. Andrews:

"The structure of every sentence is a lesson in logic."

CHAPTER 2

PROLOG

A. Overview

PROLOG was developed in the early 1970's at the University of Marseille-Aix by Alain Colmerauer and Phillipe Roussel [Roussel 75] to do natural language processing. The system evolved using ideas from Colmerauer's earlier grammar formalism, Q-systems [Colmerauer 70] and logic programming concepts (which had a long lineage including: Kowalski and van Emden; Robinson; Godel, Herbrand and Skolem; and Frege). [Robinson 83] PROLOG (which stands for PROgramming and LOGic) is an implementation of Horn clauses as a programming language with the execution mechanism provided by a depth-first, left-to-right, top-to-bottom backward chaining proof procedure that is a special case of SLD resolution. Additionally, PROLOG systems include extralogical mechanisms to control search, to manipulate program clauses and to provide some form of programming environment. [Periera 85] While PROLOG was designed for natural language processing, the fact that it uses the Horn clause subset of first order predicate logic makes it

useful as a general programming language. One of the finest examples of this is David H.D. Warren's implementation of the Dec-10 PROLOG compiler. [Periera and Warren 79] The compiler itself is written almost entirely in PROLOG. An interesting side note to this compiler is that while the compiler

1) was written while both Periera and Warren were PhD. candidates at Edinburgh,

2) 'broke new ground' in the area of logic programming, and

3) was an efficient, well written program (indeed, it's still used as a 'bench-mark'), it wasn't used by either Periera or Warren as the basis for their thesis.

In order to understand the relation of PROLOG to NLP it is necessary to have a basic understanding of how PROLOG works. With that goal in mind this section will present a simple overview of PROLOG syntax and semantics. An attempt will be made to balance theory with pragmatics and to clarify using simple examples.

B. Syntax

Appendix A contains the formal definition of the syntax of the 'standard' or Edinburgh PROLOG, presented in a Backus-Naur type notation. There are a few points of the syntax worth noting. First, words can be of any length. This capability helps make PROLOG code easy to read. Second, is the simplicity of the syntax. The syntactical simplicity coupled with the relatively small number (generally less than 150) of built-in predicates

makes the language easy to learn. (At least for people new to programming, the non-procedural nature of PROLOG can be somewhat of an enigma to programmers experienced in a more conventional language). Third, is the importance of lists. The list is the basic data structure for PROLOG, other structures (ex., matrices, trees, frames) are built by combining lists. (PROLOG shares this feature with other languages, noticibly LISP. These two languages are the most used in the area of Artificial Intelligence and Natural Language Processing. (LISP is used mainly in the United States and PROLOG elsewhere.) The list representation has been found to be adequate for most problems.) Certainly there can be no conceptual problem with representing a sentence as a list of words.

C. Semantics

There are two conceptual approaches to PROLOG semantics. The first is procedural in nature. This is the approach used by many experienced programmers that are new to PROLOG. One views each clause as a procedure definition. The conclusion of a clause is the procedure name, and the conditions of the clause represent the procedure body. However, many people with programming experience make the jump from problem to sequence of steps producing a solution without clearly formulating the facts on which their solution is based. [Davis 85]

The other viewpoint is that PROLOG has a declarative semantics. That is, when programming in PROLOG some facts and rules are asserted about individuals and their relationships. A solution is directly described rather than describing a process that results in the computation of a solution. For example, to build a database of family relationships, certain facts need to be asserted. The relationships

X is the father of Y.

Mary is the mother of Rob.

are represented by the atomic formulae

```
father(X,Y).
mother(mary,rob).
```

Similarly, continuing to build up facts about the family tree, the grandparent relationship is defined with the rules

```
grandparent(GP,GC) :- parent(GP,P), parent(P,GC).
parent(P,C) :- father(P,C).
parent(P,C) :- mother(P,C).
```

The rules are declarative in nature, that is they state that if GP is a grandparent of GC then GP is the parent of some P and P is the parent of GC. The rules don't need to define any method of finding any of GP,GC or P. Also note that there is no apriori indication which values will be known. That is the rules are "invertible". One can as easily ask

```
        ?- parent(mary,X).
```

and get the reply

```
        X = rob
```

as well as

```
        ?- parent(X,rob).
```

and get the reply

```
        X = mary
```

or even

```
        ?- parent(X,Y).
```

and get

```
        X = mary
        Y = rob
```

This invertibility is the direct result of PROLOG's declarative nature and is one of its most powerful features. Another example may show the utility of invertibility more clearly.

One of the important type of lists in LISP is the property list. It is the basis for many higher data structures including flavors, frames, etc. The general idea is that objects can have attributes or properties with associated values. For instance John may have the property 'height' with a value of 175 cm., or John may have the property 'part-of-speech' with the value of 'proper noun'. Generally in LISP one asks questions of the type

What is John's height?

or

What part-of-speech is the word John?

but asking questions of the type

Who is taller than 160 cm.?

or

What parts-of-speech are represented?

are extremely difficult to ask, using the standard property list representation. However, if the properties are represented as PROLOG relations such as

```
property(john,height,175).
property(john,part-of-speech,proper-noun).
```

Then the properties can be queried in any 'direction'. That is

Who is taller than 160 cm.?

becomes

```
?- property(Who,height,Y), Y > 160.
```

and

What parts-of-speech are represented?

becomes

```
?- setof(X,property(-,part-of-speech,X),Set).
```

That these may not be intuitive queries may be disheartening, but the mere fact that they can be asked at all should demonstrate the power of declarative programming. In actual usage, one would write a Natural Language Interface to the system to allow easier user interaction. Of course the way that is usually done in PROLOG is by using Definite Clause Grammars, which are explained in the next section.

CHAPTER 3

Definite Clause Grammar

Definite clauses are Horn clauses with a nonempty consequent. This name comes from the fact that a definite clause has a single definite conclusion, as opposed to a general clause, which can be put in a form with a disjunctive conclusion. [Periera 85]

Definite Clause Grammars (DCG) which are based on definite clauses, are an extension of Context Free Grammars. The PROLOG implementation of DCG is really a "syntactic sugar" for the actual PROLOG code, that is DCGs are not built into PROLOG but are translated into PROLOG code by the compiler. DCGs are based on difference lists, which are a powerful PROLOG technique that uses the flexibility of uninstantiated variables to limit unnecessary searching. So to better understand DCGs, let's examine the structure of difference lists.

One could write a parser by breaking up a sentence into subject, object and verb and representing those parts in the following way.

```
subject([the,big,dog|Tail],Tail).
subject([a,bluebird|Tail],Tail).
subject([that,animal|Tail],Tail).

verb([sings|Tail],Tail).
verb([bites|Tail],Tail).

object([the,mailman|Tail],Tail).
object([contatas by bach|Tail],Tail).
```

and then use the following grammar to parse possible sentences

```
sentence(Start,End) :-
   subject(Start,Point1),
   verb(Point1,Point2),
   object(Point2,End).
```

then the query

```
?- sentence([the,big,dog,bites,the,mailman],[]).
```

would succeed in the following way.

Sentence would begin by trying to find a subject; (i.e., subject would be called in the form

subject([the,big,dog,bites,the,mailman],Point1))

'subject' would match [the,big,dog|Tail] and unify the tail of the sentence (i.e., [bites,the,mailman]) to the variable Point1. (Note that the list [bites,the,mailman] is the difference between the complete sentence and the part that 'subject' matches (i.e., [the,big,dog]) thus the name difference list. This also is the realization of the afforementioned flexibility of uninstantiated variables.)

Verb would then match [bites|Tail] and unify the rest of the sentence to Point2. Finally, object would match ([the,mailman|Tail],[]) (the variable End was instantiated to [] in the query) and PROLOG would reply, yes. In effect saying [the,big,dog,bites,the,mailman] is a sentence. It is obvious that for small grammars this is fast and efficient but lacks enough power to be truly useful as it's implemented here.

DCGs add more power and flexibility than this and also add a "sweeter" way of writing it. The "syntactic sugar" comes in with the way DCGs are actually written. A more flexible grammar, than the above, could be written in DCGs as

```
s --> np, vp.
np --> det, n.
np --> det, adj, n.
vp --> v-intrans.
vp --> v-trans, np.
det --> [the].
adj --> [big].
n --> [dog].
n --> [mailman].
v-trans --> [bites].
v-intrans --> [sleeps].
```

This would be translated by the compiler into the following PROLOG code.

```
s(Head,Tail) :- np(Head,Middle), vp(Middle,Tail).

np(Head,Tail) :- det(Head,Middle), n(Middle,Tail).
np(Head,Tail) :- det(Head,Mid1), adj(Mid1,Mid2),
                 n(Mid2,Tail).

vp(Head,Tail) :- v-intrans(Head,Tail).
vp(Head,Tail) :- v-trans(Head,Middle), np(Middle,Tail).

det([the|Tail],Tail).
```

```
adj([big|Tail],Tail).

n([dog|Tail],Tail).
n([mailman|Tail],Tail).

v-trans([bites|Tail],Tail).

v-intrans([sleeps|Tail],Tail).
```

The DCG notation is obviously much more readable, much easier to write and much easier to maintain and modify. Also there is a much closer relationship between the DCG and the Backus-Naur Form given in Appendix A. This similarity suggests that we can use DCGs to implement simple syntax checkers for languages for which we have a Backus-Naur description. [Boisen 87]

While the DCG formalism so far presented is quite powerful there are a number of extensions which have been added to it to make it even stronger. First, regular PROLOG terms can be included in the grammar if they are enclosed in curly braces. Such clauses are not translated but simply inserted into the resulting PROLOG code. This can be used to separate the dictionary (lexicon) from the grammar so that 'n --> [bat].' doesn't have to be written every time a noun is introduced, but could instead be translated directly into DCGs. Another useful feature is the ability to add an extra argument to the symbols in the DCG notation. These can be used for checking agreement between constituents, tracking context or other bookkeeping tasks. [Boisen 87]

From a linguistic viewpoint, one of the nicest features of the DCG formalism is that the grammar is executable, this isn't the case for other parsers. In a LISP implementation of Augmented Transition Networks (ATN) [Woods 72] for instance, the grammar and the executable code aren't even in a one-to-one relationship much less the same code. The major criticism of DCGs from its detractors is that all DCGs are in effect an implementation of a context-free grammar and lack the sophistication for really 'robust' natural language systems. Yet it has been shown that other linguistic grammar formalisms can easily be implemented with the Definite Clause Grammar notation [ATNs:Periera 80, Montague Grammar:Jowsey 86, Unification-based Grammar:Hirsch 87, Word Expert Parser:Papegaaij 86]. (Actually Periera did more than just show that ATNs were implementable in DCGs. He also demonstrated that DCGs are more powerful than ATNs and have the advantages of modularity, perspicuity and efficiency over them). It is also worth noting that it is often necessary to backtrack when parsing ambiguous sentences and backtracking is expensive in ATN directed parsers. [Kalish 87] However, the DCG formalism isn't the final answer to NLP, just a step along the way.

CHAPTER 4

CHAT-80

In the early 1980's, Fernando Periera wrote "Logic for Natural Language Analysis" as a thesis submitted to the Department of Artificial Intelligence, University of Edinburgh for the degree of Doctor of Philosophy. [Periera 83] This thesis is significant for a number of reasons. It was theoretically interesting in that the basis for the NLP system presented was Extraposition Grammar (XG). XGs are an extension of Definite Clause Grammar designed to handle a problem found frequently in relative clauses, namely left extraposition. Left extraposition occurs when some part of a sentence is completed by a part that previously occurred (is to the left of the original part) in the sentence. For example, in the sentence:

The country that borders Korea has a tonal language.

The clause 'that borders Korea' is completed by the noun phrase 'The country' but the noun phrase preceeded it in the sentence.

Extraposition Grammars solves this ambiguity by placing markers in the relative clauses, a sort of relative pronoun variable, and binding them within the noun clause in which they occur.

As a demonstration of his ideas, Periera wrote the Chat-80 NLP system. Chat is a natural language front-end for a global geographical data base. (Actually, there are two distinct parts to Chat, Periera's NLP front-end and David H.D. Warren's query optimization techniques.) Since Chat was the first serious attempt at using Definite Clause Grammars for NLP and since it was available to a large number of people for experimentation, Chat's position in the NLP community is an important one. Its premier position may in fact be a mixed blessing. While it has positively influenced most of the succeeding logic programming NLP systems, its shortcomings have been used as demonstrations of PROLOG's unsuitability for NLP systems!; a strange position to take with a programming language designed for NLP and seems to be a case of 'throwing the baby out with the bath water'. With that as an introduction, let's examine Chat more closely.

As might be expected Chat does very well with relative clauses. It can answer questions as complicated as:

> Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India?

The query is first syntactically parsed and categorized as being one of four basic type of queries: declarative, yes/no, wh-type, or imperative. This parse is then sent to the semantic processor where it is checked semantically, and if meaningful, converted to a query. The query is then mathematically optimized for efficient access to the database, the database is searched for an answer and the answer is printed.

Of course, Chat was designed as a demonstration program and therefore doesn't meet many of the requirements for a 'robust' NLP system. It handles one sentence at a time (i.e., query-answer) and thus has no dialog capabilities. It cannot handle anaphora, ellipses or sentence fragments. It cannot handle general conjunction though it does handle the conjunction of two relative clauses. It has a very user-UNfriendly interface, gives very little feedback on failures, and degrades awkwardly.

Many of these shortcomings were recognized by Periera at the time Chat was written. In fact, in Chapter 6 of his thesis [Periera 83], he discusses a number of them. The first of these is conjunction. Theoretically the interaction of left extraposition and conjunction causes a problem. For example, in the phrase

the letter that Mary wrote and sent

the noun phrase would have to be extraposed for both of the conjoined verb phrases which is not possible with XGs. He proposed a way of handling this problem with 'conjunction schema' which use an abstract variable for the marker but decided that he didn't want to tackle the problem in Chat.

The second shortcoming discussed is the reliance on PROLOG's top-down approach for his parser. While a top-down approach is more efficient for well-formed input, it has difficulty in handling sentence fragments or allowing error correction. Practical dialog systems require both capabilities. (PROLOG's top-down, depth-first, control procedure has drawn unfair criticism from people unfamiliar with PROLOG's true power. Actually the top-down control is provided gratis by PROLOG, if you want some other control scheme (ex., breadth-first, best-first) it's as easy to write it in PROLOG as it is in LISP. The difference is that in LISP you have to write something and in PROLOG you don't). A number of systems since Chat have addressed these shortcomings, in particular, the BUP system described in the next section can handle sentence fragments, and the CHARISMA system deals, though in a somewhat ad hoc manner, with error correction [McHale 87].

Periera also states that the organization of his dictionary would be a problem for anyone moving to a new domain. Certainly it is organized a bit strangely. Accordingly to Chat, every verb

is a regular verb and one has to add each verb in half-a-dozen different places. Rather than entering verbs as: irregular-verb(present-tense, past-tense, future-tense, ...) one must add each verb to the present-tense clauses, the past-tense clauses, etc. Notice the last sentence says 'each verb'. The program doesn't take care of the morphological changes for the truly regular verbs but instead requires the dictionary builder to add them explicitly. [This problem was also addressed in CHARISMA]. Changes to the semantic hierarchy and the world rule-base aren't as complex but the necessity of changing all three is problematic for those wishing to modify Chat.

The fourth shortcoming discussed is the relationship between quantification and referents. Periera uses classical quantifiers as the exclusive representation of bindings but feels that in order to handle anaphora, or other structures of dialog, a more flexible notion of bindings would be necessary.

Chat-80 then is an important milestone in the short history of NLP. It not only demonstrated the utility of Definite Clause Grammars and PROLOG in general but also paved the way for future research. One of the systems that Chat set the stage for was the Bottom Up Parser.

CHAPTER 5

Bottom Up Parser (BUP)

Shortly after Periera produced the Chat-80 system, Yuji Matsumoto and Hozumi Tanaka wrote BUP, A Bottom-up Parser Embedded in PROLOG [Matsumoto 83]. The abstract of BUP states the reason BUP was written was to maintain the perspicuity, power, generality, et al, of DCG while allowing the grammarian to write a grammar with left recursive rules. (Which isn't allowed in DCGs that rely on PROLOG's top-down control mechanism). Other features of the system include a greater separation of the dictionary from the grammar, some morphological treatment of words, the handling of some idiomatic expressions and various tools to aid the grammarian in developing an NLP system. The system was originally written to parse Japanese but has since been expanded to handle English.

The parser that BUP uses is a bottom up, left corner parser with top-down expectation. BUP is divided into three parts: goals, rules and dictionary. While the dictionary will be the largest part for any real implementation, the goal and rule part

define BUPs actions, as the dictionary is only called from goal. From a control viewpoint, goal could be written as:

```
goal(Goal,List,Tail) :-
   dictionary(PartOfSpeech,List,NewTail),
   PartOfSpeech(Goal,NewTail,Tail).
```

Of course, this isn't executable PROLOG, but the idea is that goal looks in the dictionary for the part of speech of the first word in the list. Then the proper rule for that part of speech is called which in turn either matches a terminal or calls goal with a new subgoal.

To write a grammar in BUP, one uses DCGs. This style was maintained for the reasons listed above. BUP then translates the DCG into BUP rules and the dictionary. [Matsumoto 83] gives numerous examples of translations of DCG into BUP code, while [Matsumoto 85] deals more with idiomatic expressions.

BUP has included some nice extensions to the DCG type grammar that Chat uses. Some of these are a linking mechanism that is a local top-down mechanism that checks appropriateness of rules and thus speeds goal selection; an ability to do some morphological analysis of words thus simplifying the task of the system builder (a feature that should be deemed necessary for NLP systems that deal with inflectional languages); a way of limiting backtracking by storing successes and failures; and the ability to handle some sentence fragments.

The point being stressed about BUP is not that it is a 'robust', end user ready system but that it demonstrates how relatively easy it is to modify and extend the DCG formalism. This particular system has shown that it is possible to give a different control structure in PROLOG. Furthermore once a DCG type grammar is written for one natural language it is flexible enough to be extended for a different language with minimal effort. In regards to robustness Periera states in the introduction to "PROLOG and Natural Language Analysis",

> "One of the major insufficiencies remaining in the text is a lack of linguistic sophistication and coverage evinced by the analyses we use. The reader should not think that such naivete inheres in PROLOG as a tool for natural language analysis...".[Periera 87]

CHAPTER 6

Modular Logic Grammars (MLG)

Modular Logic Grammar is not a descendant of Chat but Chat has influenced the design and implementation of MLGs. Michael McCord has developed the MLG formalism to accomplish a number of goals. The most important of these, seems to be the desire to be able to mix syntactic and semantic analysis. As Richard Kittredge observes:

> "MLGs are syntactically similar to DCGs, but with distinctions between strong and weak non-terminals, to help separate grammatical categories with semantic import and those which are used as auxiliaries during treatment of non-compositional structures. There are also logical terminals, used to build up pieces of semantic representation. Compiled MLG rules may apply in single-pass mode, where calls to semantics are interleaved with application of syntactic rules, giving only semantic (logical) forms as the output, or they may apply in two-pass mode to build first a syntactic structure which is passed to the semantic interpreter." [Kittredge 87]

MLGs also allow left recursive grammar rules but handle them quite differently from BUP. Rather than doing a bottom-up parse, MLG still maintains a top-down parse but flags left-recursive

rules to be rewritten by the MLG compiler. MLGs can also handle left extraposition but do so by using explicit topic-pair arguments as opposed to Periera's XGs; this also allows for easier handling of conjunctions. At least it removes the theoretical conflict that occurs in XGs between the scoping of conjunctions and left extraposition. Another point of interest is McCord's structuring of the lexicon. He assumes the lexicon is too large to fit into main memory and must reside on disk. He gives an algorithm for fast lexicon searching and thus addresses the problem of 'scaling-up' that many systems are criticized for ignoring. The lexicon is somewhat more cohesive than Chat's and allows a morphological rule system to infer the correct tense forms for regular verbs. Both of these are welcome extensions to the Chat methodology.

CHAPTER 7

Zen, Haiku and PROLOG

The intent of this chapter is two fold. First, the connection of PROLOG with the Japanese Fifth Generation Computer Project will be presented and second, those properties that PROLOG and Zen have in common will be examined.

A. Fifth Generation Computing

While LISP is the favorite language of Artificial Intelligence (AI) researchers in the United States, PROLOG is the Lingua Franca for the rest of the AI world. A paramount example of this is PROLOG's part in the Japanese Fifth Generation Computer Systems (FGCS) project. Logic programming is envisioned in the FGCS as the missing link unifying the various fields of computer science and is thus highly emphasized. The reasons for this include logic programming's suitability for: problem specification, relational databases and query languages, rule-based expert systems and natural language processing. More importantly perhaps, is the view that logic programming is much

better suited to parallel processing than any other programming paradigm. [Fuchi 83] It isn't anticipated that PROLOG will replace LISP in American AI research, but it seems very narrow minded at best to ignore PROLOG in places where it out performs LISP and both parallel processing and natural language processing may well be such domains.

B. Zen in the Art of PROLOG

At one level Zen and PROLOG can be viewed as being diametrically opposed. That is, Zen states that the only way to attain an understanding of truth is not through logic but through experiencing 'what is'. The Zen experience is to cut through the surface structure of objects and discover their basic relationships and functions. It can be seen as the ultimate honing of Occam's razor, that is, "cut away the superfluous and what is left is reality".

A realization of the spirit of Zen can be found in haiku. Haiku are short poems that strive for beauty and elegance within the bounds of a highly constrained structure. In the best written PROLOG programs (certainly a subjective judgement), there is a striving for logical elegance that is intuitively similar to the creation of haiku. Perhaps this "intuitive similarity" isn't obvious to everyone, certainly the Zen masters had no intention of comparing haiku to PROLOG, yet the fit seems so natural. Periera seems to say something similar,

> "But PROLOG might be most easily learned by ignoring previous experience with other programming languages and trying to absorb the PROLOG gestalt from first principles." [Periera 87].

Of course, the comparison of PROLOG to haiku could be viewed differently. The question could be one of 'can a PROLOG program be written that writes haiku?'. I feel the answer to that is yes and no. Certainly one could write a program which follows the rules of construction of haiku, (i.e., 17 syllables, references to time, season, etc.) but presently it is not possible to write a program that consistently produces meaningful haiku. This may be seen as a result of lack of intelligence in AI systems, though it is doubtful that most people could write more meaningful haiku than present programs can.

The whole area of natural language output has been ignored in this paper. This shouldn't be judged as an indication of PROLOG's inability in this area, but rather as an indication of the author's familiarity with the input problem. For those interested in output, an interesting section on generation of poetry, including haiku, can be found in the book by Goldenberg [Goldenberg 87]. This book deals with the computer language Logo, which is closer to LISP than PROLOG, but the translation of non-graphic Logo routines to PROLOG is rather straightforward.

CHAPTER 8

SUMMARY

The intent of this report was to show the strong relationship between PROLOG and Natural Language Processing. The growth of PROLOG has been closely tied to its use as a programming paradigm for NLP. While the future of logic programming may be more closely related to hardware considerations (i.e., parallel processing), the continued reliance on logic programming for NLP will emphasize those aspects that best suit both purposes (the Word Expert Parser of Papegaaij is inherently parallel-izable).

No real attempt at explaining the details of either the logic or syntax of PROLOG has been made. Those interested in questions of logic should see [Lloyd 81] or similar texts. The best texts on PROLOG are Clocksin and Mellish [Clocksin 81] and Sterling and Shapiro [Sterling 86]. The latter is highly recommended.

The contents of this report are due to me with one reservation. I feel that the only way most of us have 'original' ideas is by combining the thoughts of others in new ways, thus this report is a product of my experience and exposure to the thoughts of others. I have made an attempt to footnote all the places where I could pinpoint someone else's idea, all oversights are unintentional.

APPENDIX A

PROLOG SYNTAX

```
<clause> ::= <atmf> "." | <atmf> ":-" <atmfs> "." |
             ":-" <atmfs> "."

<atmfs> ::= <atmf> ("," <atmf>)*

<atmf> ::= <predicate> "(" <terms> ")"

<terms> ::= <term> ("," <term>)*

<term> ::= <variable> | <constant> |
           <function> "(" <terms> ")" | <list>

<variable> ::= (uppercase-letter | "-") <word>

<word> ::= (letter | digit)*

<constant> ::= number | <lowerword>

<function> ::= <lowerword>

<predicate> ::= <lowerword>

<lowerword> ::= lowercase-letter <word>

<list> ::= "[]" | "[" <head> "|" <tail> "]"

<head> ::= <term>

<tail> ::= <list>
```

(where '<atmf>' is an atomic formula,
'|' is a disjunction,
sentential elements are quoted and
'*' allows unbounded repetition) [Davis 85]

BIBLIOGRAPHY

[Boisen 87] Boisen, S.
Language Processing Using Definite Clause Grammars
AI Expert, 46-56, June 1987

[Clocksin 81] Clocksin, W.F., Mellish, C.S.
Programming in PROLOG
Springer-Verlag, 1981, 1984

[Colmerauer 70] Colmerauer, A.
Les Systems-Q ou un Formalisme pour Analyser et Synthetiser des Phrases sur Ordinateur
Internal Publication 43, Department d'Informatique, Universite de Montreal, Canada, 1970

[Davis 85] Davis, R.E.
Logic Programming and PROLOG: A Tutorial
IEEE Software, 53-62, September 1985

[Fuchi 83] Fuchi, K.
The Direction the FGCS Project Will Take
New Generation Computing, 1(1):3-9, 1983

[Goldenberg 87] Goldenberg, E.P., Fuerzeig, W.
Exploring Language with Logo
MIT Press, 1987

[Herrigel 53] Herrigel,E.
Zen in the Art of Archery
Random House, New York, 1971 (Reprint)

[Hirsch 87] Hirsch, S.B.
P-PATR: A Compiler for Unification-Based Grammars
Proceedings of Second International Workshop on NL Understanding and Logic Programming, 63-74, Simon Fraser University, August, 1987

[Jowsey 86] Jowsey, H.E.
Montague Grammar and First-Order Logic
DAI Working Paper No. 190,
University of Edinburgh, 1986

[Kalish 87] Kalish, C.
A Portable Natural Language Interface
RADC-TR-87-155, September 1987

[Kittredge 87] Kittredge, R.
Advanced Command and Control Environment,
Natural Language Interface Investigation
Final Report, RADC, 1987

[Lloyd 84] Lloyd, J.W.
Foundations of Logic Programming
Springer-Verlag, 1984

[Matsumoto 83] Matsumoto, Y., Tanaka, H.
BUP: A Bottom-Up Parser Embedded in PROLOG
New Generation Computing, 1(2):144-158, 1983

[Matsumoto 85] Matsumoto, Y., Kiyono, M., Tanaka, H.
Facilities of the BUP Parsing System
NL Understanding and Logic Programming
Elsevier Science Publishers B.V.,
North-Holland, 1985

[McCord 86] Walker, A., McCord, M., Sowa,
J.F., Wilson, W.G. (eds)
Knowledge Systems and PROLOG:
A Logical Approach to Expert Systems and
Natural Language Processing
Addison-Wesley, 1987

[McHale 87] McHale, M.L., Huntley, M.A.
CHARISMA
Internal Report, RADC, August, 1987

[Papegaaij 86] Papegaaij, B.C., Sadler, V., Witkam, A.P.M. (eds)
Word Expert Semantics:
an Interlingual Knowledge-Based Approach
BSO, Netherlands, 1986

[Periera 79] Periera, F.C.N., Warren, D.H.D.
User's Guide to DECsystem-10 PROLOG
Occasional Paper 15,
Dept. of Artificial Intelligence,
University of Edinburgh, 1979

[Periera 80] Periera, F.C.N., Warren, D.H.D.
Definite Clause Grammars for Language Analysis
Artificial Intelligence, 13:231-278, 1980

[Periera 83] Periera, F.C.N.
Logic for Natural Language Analysis
Technical Note No. 275, SRI, Menlo Park, CA, 1983

[Periera 85] Periera, F.C.N.
PROLOG with Natural-Language Examples
Presented as a tutorial at the
23rd Annual Meeting of the
Association for Computational Linguistics,
Chicago, 1985

[Periera 87] Periera, F.C.N., Shieber, S.M.
PROLOG and Natural-Language Analysis
CSLI, Stanford University, 1987

[Robinson 83] Robinson, J.A.
Logic Programming-Past, Present and Future
New Generation Computing, 1(2):107-124, 1983

[Roussel 75] Roussel, P.
PROLOG: Manuel de Reference et Utilisation
Technical Report, Groupe d'Intelligence Artificelle,
U.E.R. de Luminy,
Universite d'Aix-Marseille II, 1975.

[Sterling 86] Sterling, L., Shapiro, E.
The Art of PROLOG: Advanced Programming Techniques
MIT Press, 1986

[Woods 72] Woods, W.A., Kaplan, R.M., and Nash-Webber, B.
The Lunar Sciences Natural Language
Information System: Final Report.
Report 3438, BBN Inc., June, 1972.

U.S. GOVERNMENT PRINTING OFFICE: 1988—511-117/64067

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.